Towards An Assume-Guarantee Theory for Adaptable Systems

Paola Inverardi, Patrizio Pelliccione, Massimo Tivoli
Dipartimento di Informatica
Università degli studi dell'Aquila
{inverard,pellicci,tivoli}@di.univaq.it

Abstract

Modern software systems should be more and more designed with adaptation and run-time evolution in mind. But even with good reactions to changes, the triggered adaptation should be performed preserving some properties that we call invariants*. This position paper presents a step towards the definition of a theoretical assume-guarantee framework that allows one to efficiently define under which conditions adaptation can be performed by still preserving the desired invariant. The framework aims to cope with different levels of granularity that span from code to software architecture. Two illustrative examples instantiate the framework at two different levels of abstraction.*

1. Introduction

Increasingly, software systems must adapt in response to "*changing user needs, system intrusions or faults, changing operational environment, and resource variability*" [3]. Systems should be flexible enough to allow the introduction of new functionalities incrementally, both before and after system deployment. Therefore the software system must be designed with run-time evolution in mind and, at the same time, solutions must be adopted to keep the user's perceived as well as system intrinsic dependability at a satisfactory level. Mechanisms for run-time evolution are needed to provide good reactions to both system and context changes. But even with good reactions, a loss in dependability may make the system unusable in practice. The ability to effectively react to changes without degrading "*a set of high-level goals*" [3] is the key factor for delivering successful eternal systems that continuously satisfy evolving user requirements. Hereafter we call *invariants* these high-level goals since they represent the system properties that "*should be maintained regardless of the environmental conditions*" [3]. Instead, non-critical properties might be relaxed, hence increasing the degree of flexibility of the system during or after adaptation [3].

In this context, there is the need of theories, methodologies and techniques able to effectively support adaptation, and the consequent evolution, of the system at run-time. Furthermore, adaptation can happen in different forms namely (self-)adaptiveness/dynamicity and at different levels of granularity, from software architecture to line of code [3]. Therefore, a general framework should allow one to describe adaptation in terms of changes in some *parts* of the system that once applied preserve an *invariant* and should cope with different levels of granularity.

In this position paper we propose a theoretical assume-guarantee framework that allows us to break up a system S in parts that can be substituted/changed without hurting the invariant property. The framework allows the definition of efficient conditions, on the parts affected by the change, to be proved at run-time for guaranteeing the correctness of the adaptation. We consider different types of *adaptation*, i.e., addition, removal, substitution of system's parts, and changes in the environment, applied at different levels of abstraction spanning from code to software architecture.

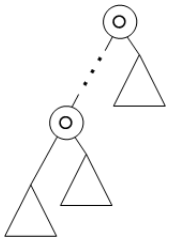

Figure 1. Abstract tree

We assume that S is built by using some concrete syntax, thus the abstraction used in the following is to identify a system with a syntax graph. For the purpose of this paper it is enough to consider a very simple syntax that composes systems through a composition operator $\circ$ that also represents the "breaking" point of adaptation, hence we abstract from concrete operators. Thus we can assume that our system is represented through an abstract tree whose nodes are either system components (leaf nodes) or compositional op-

erators (intermediate nodes), see Figure 1. Actually, the abstract tree is hierarchical in the sense that system component nodes can in turn contain abstract subtrees.

Adaptation can occur at each system component node and also at its sub-components. The idea is to label every node with an assumption to be satisfied by the component in order to maintain the invariant. These assumptions should be automatically generated by following a compositional approach, e.g., by using the approach presented in [5] when this applies.

The main motivations of the framework preliminarily described in this paper, are as follows.

• *To ease the task of effectively breaking the system into parts* in order to perform adaptation and prove its correctness (with respect to the invariant) by following a compositional approach that provides an advantage in memory and time consumption with respect to monolithic verification. This would support the run-time adaptation and evolution of the system.

• *To ease the task of correctly (with respect to the invariant) composing a system out of elementary components*. For each sub-system (an elementary component or a more complex component), a pair of assume and guarantee properties is calculated hence providing the sub-system with an abstract specification of all the possible programs (contexts) where it can be used as a sub-program. This would support design-by-contract approaches to system development.

• *To support adaptability at different levels of system granularity and at different stages (before and after deployment).*

• *Compositional assumption generation*. Assumptions of the system are generated by following a compositional approach driven by the assume-guarantee strategy. This allows the detection of "adaptability points" that are the system parts that can be adapted and, hence, affected by a possible change. The assumption generation is one of the most expensive task of the assume-guarantee reasoning and, by performing it in a compositional fashion, our framework would mitigate its complexity.

• *To support reactions to unsuccessful (with respect to the invariant) adaptations*. When a change occurs and it triggers an adaptation invalidating the invariant, our framework can either (i) guide the developer to localize the problem and to find a solution (e.g., replacing/removing/adding components) or (ii) automatically refine the assumption generated for the part of the system affected by the change, when possible. In other words, in general, our framework supports also the automatic synthesis of the context providing the system with what is needed to preserve the invariant.

This paper is organized as follows: Section 2 provides a brief introduction to assume-guarantee reasoning required to understand the framework that is presented in Section 3. Section 4 presents two examples that instantiate the framework at two different levels of abstraction: code, presented in Section 4.1, and software architecture, presented in Section 4.2. Related work is presented in Section 5. In Section 6, we discuss the fundamental aspects of our approach and summarize possible application scenarios. This paper concludes in Section 7 providing insights on future research directions.

2. Assume-guarantee reasoning

Assume-guarantee reasoning [10, 15] is a technique that, by considering a system composed of several components, aims at verifying the system through the separate verification of the single components. More precisely, each single component cannot be considered isolate but behaving and interacting with its context (i.e., the rest of the system). Actually, in assume guarantee, the context is represented as a property that the context should satisfy to correctly interact with the component. This property is called *assumption* meaning that it is the assumption that the component makes on the context. If this assumption is satisfied by the context, the component that behaves in this context will satisfy another property, called *guarantee*. By combining assume/guarantee properties in an appropriate way, it is possible to demonstrate the correctness of the entire system without constructing the complete system proof. Several works propose suitable decompositions of the system (e.g., [1]) and automatic generation of the assumptions (e.g., [5]).

The notation used in [15] by Pnueli is the following:

$$< \varphi > M < \psi >$$

The common reading is "if the context of M satisfies φ, then M in this context satisfies ψ". Below we show the classical reasoning chain:

$$\frac{<> M' < \varphi > \quad < \varphi > M < \psi >}{<> M \cdot M' < \psi >}$$

Its interpretation is: if M' satisfies φ and M, over a context that satisfies φ, satisfies ψ then $M \cdot M'$ will satisfies ψ, where "$\cdot$" is a suitable composition operator.

3. Assume-Guarantee Framework for Adaptable systems

An adaptable system S can be seen as the composition V$\circ$C, through an abstract operator "$\circ$", of a core part C and an adaptable part V such that S satisfies the invariant I (S$\models$I). The semantics of $\circ$ is established by the operational semantics of the language of S that has to be considered as one of the main ingredients of our theory. Furthermore, as

it would be clear later, the operational semantics of the language of S is crucial also to dictate how to orchestrate the compositional assumption generation process. I is a safety property of S. In this position paper, we focus only on safety properties since, in general, the existing approaches for automatic assumption generation (see [2, 5, 6, 8] and reference therein) are limited to such kind of properties. Furthermore, I should be expressed in a way that it is possible to reduce its verification to a compositional check as, e.g., it is shown in [2] where the deadlock detection problem is reduced to a trace containment compositional check. In particular, I can be expressed in terms of patterns of events (interactions) and/or state evaluations (e.g., values of variables or current mode of operation) and their relative ordering over time. C, as a core part, cannot be affected by a change. Instead V, as a variable part, is affected by a change, i.e., it is the part of the system where adaptation is performed. Neither C nor V are fixed, since C and V are identified by the particular adaptation to be performed.

To enable compositional reasoning, the $\circ$ operator must enjoy the associativity property:

$$M_1 \circ (M_2 \circ M_3) \equiv (M_1 \circ M_2) \circ M_3 \equiv M_1 \circ M_2 \circ M_3$$

where $\equiv$ is a behavioral equivalence relation with respect to the behavior of the composed system obtained through the $\circ$ operator. If $\circ$ is commutative, then the same system S can be composed in different ways. This implies that we can have different sets of assume-guarantee annotations for the same system. However, note that the global invariant I and the assumptions on the context of S required to satisfy I are always the same. What can be different are only the local/internal assumptions and guarantees of the system's parts.

S lives in a possibly empty context E. That is either S is closed (E is empty) and I characterizes the requirements always guaranteed by S or S is open (E is not empty) and I characterizes the assumptions of S on E in order to satisfy I. Let S be composed of n components[1] $\mathrm{M_1}$,$\mathrm{M_2}$,$\cdots$,$\mathrm{M_n}$ through the $\circ$ operator: $\mathrm{S}=\mathrm{M_1}\circ\mathrm{M_2}\circ\cdots\circ\mathrm{M_n}$.

The idea is to produce for each $\mathrm{M_i}$ ($1<i<n$), $\mathrm{G_i}$ and $\mathrm{A_i}$ representing respectively the local invariant that $\mathrm{M_i}$ has to guarantee and the assumptions that characterizes the context in that point. In other words, $\mathrm{M_i}$ will guarantee $\mathrm{G_i}$ when its context satisfies $\mathrm{A_i}$. Let us assume that $\mathrm{M_i}$ is the component that is affected by a change, thus $\mathrm{V}=\mathrm{M_i}$. If i=2,$\cdots$,$n-1$, thanks to the associativity property of $\circ$, C can be decomposed into $\mathrm{C_i^{left}}=\mathrm{M_1}\circ\cdots\circ\mathrm{M_{i-1}}$ and $\mathrm{C_i^{right}}=\mathrm{M_{i+1}}\circ\cdots\circ\mathrm{M_n}$. Instead, if i=1 (resp., i=n) then $\mathrm{C_i^{left}}$ (resp., $\mathrm{C_i^{right}}$) does not exist. In other words, $\mathrm{C_i^{left}}$ is the context of $\mathrm{M_i}$ while $\mathrm{C_i^{left}}\circ\mathrm{M_i}$ is the context of $\mathrm{C_i^{right}}$ (when both $\mathrm{C_i^{left}}$ and $\mathrm{C_i^{right}}$ exist). Otherwise if $\mathrm{C_i^{left}}$ (resp., $\mathrm{C_i^{right}}$) does not exist, then $\mathrm{M_i}$ has no "left context" (resp., "right context"). For each $\mathrm{M_i}$, $\mathrm{C_i^{left}}$ and $\mathrm{C_i^{right}}$ are identified by means of the abstract syntax tree associated to the system S. For instance, let S be represented by the abstract syntax tree AT in Figure 2. AT contains two different kinds of nodes: the circle-nodes that contain the operator $\circ$ and the triangle-nodes that contain the parts S is composed of. The idea is that the adaptation is guided by $\circ$. Once a composition has been defined (for instance in figure $\mathrm{M_1}\circ\mathrm{M_2}\circ\cdots\circ\mathrm{M_n}$), $\mathrm{C_i^{left}}$ is identified by the left part of the composition with respect to $\mathrm{M_i}$. That is the parent node of $\mathrm{M_i}$ is a circle-node and its left subtree identifies $\mathrm{C_i^{left}}$. Analogously $\mathrm{C_i^{right}}$ is identified by the right part of the composition with respect to $\mathrm{M_i}$, i.e., the remaining of AT. Note that the parent node of $\mathrm{M_1}$ (resp., $\mathrm{M_n}$) has no left (resp., right) subtree.

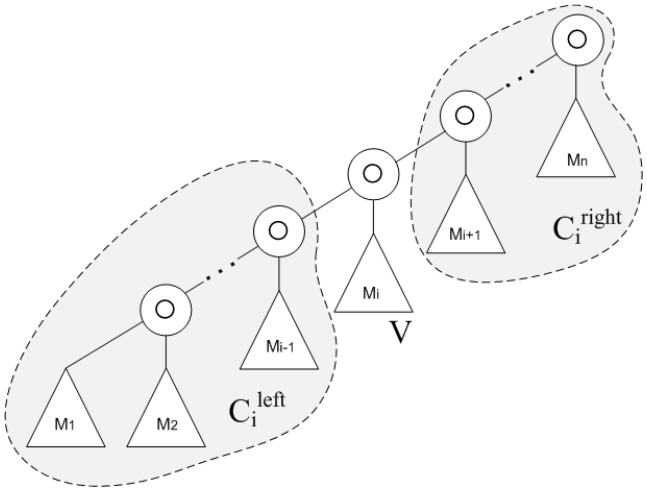

Figure 2. Abstract syntax tree of S

The $\mathrm{A_i}$ generation aiming at guaranteeing the correctness of the adaptation performed on V with respect to I, and for a system composed of n components, $\mathrm{M_1},\cdots,\mathrm{M_n}$, is regulated by the following assume-guarantee rules:

$$(1)\quad \frac{\langle\phi_E\rangle\ \mathrm{M_1}\ \langle\mathrm{G_1}\rangle \qquad \langle\mathrm{G_1}\rangle\ \mathrm{C_1^{right}}\ \langle\mathrm{I}\rangle}{\langle\phi_E\rangle\ \mathrm{M_1}\circ\mathrm{C_1^{right}}\ \langle\mathrm{I}\rangle}$$

$$(2)\quad \frac{i=2,\cdots,n-1 \qquad \langle\phi_E\rangle\ \mathrm{C_i^{left}}\ \langle\mathrm{A_i}\rangle \qquad \langle\mathrm{A_i}\rangle\ \mathrm{M_i}\ \langle\mathrm{G_i}\rangle \qquad \langle\mathrm{G_i}\rangle\ \mathrm{C_i^{right}}\ \langle\mathrm{I}\rangle}{\langle\phi_E\rangle\ \mathrm{C_i^{left}}\circ\mathrm{M_i}\circ\mathrm{C_i^{right}}\ \langle\mathrm{I}\rangle}$$

[1] Here the term component is simply used as a synonym of "part".

$$(3)\quad \frac{\langle \phi_E \rangle\ \mathrm{C}_\mathrm{n}^{\mathrm{left}}\ \langle \mathrm{A_n} \rangle \quad \langle \mathrm{A_n} \rangle\ \mathrm{M_n}\ \langle \mathrm{I} \rangle}{\langle \phi_E \rangle\ \mathrm{C}_\mathrm{n}^{\mathrm{left}} \circ \mathrm{M_n}\ \langle \mathrm{I} \rangle}$$

Without loss of generality and by considering only the rule (2), since the invariant I is known and S satisfies I, the rule suggests a backward reasoning chain that starting from I and $\mathrm{C}_\mathrm{i}^{\mathrm{right}}$ tries to generate $\mathrm{G_i}$. In turn, $\mathrm{M_i}$ guarantees $\mathrm{G_i}$ under the assumption $\mathrm{A_i}$ on its context $\mathrm{C}_\mathrm{i}^{\mathrm{left}}$. Following the reasoning we obtain ϕ_E that represents the assumption that S makes on its context. Note that ϕ_E=$true$ means that no assumption is made on the context (e.g., when S is a closed application).

In general, for each triangle-node $\mathrm{M_x}$, in order to automatically produce the assumption $\mathrm{A_x}$, our calculus decorates each $\mathrm{M_x}$ with the partial proofs that should be performed, to preserve the invariant I, when $\mathrm{M_x}$ is considered as the variable part of S. The assumptions can be automatically calculated by a suitable assumption generation function defined on the considered language. For instance, in Section 4.1, this generation function is a propositional logic formulae solver, e.g., [14], whereas, in Section 4.2, it is the $\mathcal{L}^*$ learning algorithm defined in [8].

Note that if we replace an existing $\mathrm{M_x}$ with a new $\mathrm{M'_x}$ we have to check $\mathrm{A'_x}$ against $\mathrm{A_x}$. If $\mathrm{A'_x}$ is weaker then the invariant is satisfied and hence no assumption (re-)generation is required for $\mathrm{C}_\mathrm{x}^{\mathrm{left}}$. Otherwise, assumption backtracking (re-)generation is required for $\mathrm{C}_\mathrm{x}^{\mathrm{left}}$ until we generate a new assumption that is weaker than the old one. If we reach the root of the syntax tree with a "$null$" assumption, then the applied adaption (the replacement of $\mathrm{M_x}$ with $\mathrm{M'_x}$) is not valid since an environment preserving the invariant does not exist.

A triangle-node, e.g., $\mathrm{M_j}$, can be in turn composed of subparts $\mathrm{S_{j1}},\cdots,\mathrm{S_{jk}}$ by means of a $\bullet$ operator as shown in Figure 3. The compositional assumption generation process on $\mathrm{M_j}$ depends on the assumptions generated for each $\mathrm{S_{j1}},\cdots,\mathrm{S_{jk}}$ and it is orchestrated accordingly to the operational semantics of the operator $\bullet$. The $\bullet$ operator can be a specific operator of the language (e.g., a **if-then-else-fi** or a **while-do-endw** control flow statement in an imperative programming language). Each $\mathrm{S_{j1}},\cdots,\mathrm{S_{jk}}$ is an abstract syntax (sub-)tree. As shown in Figure 3, each $\mathrm{S_{ji}}$ can in turn be built through the $\circ$ operator hence leading to a hierarchical structure.

Let $\mathrm{S_{ji}}$ be composed of $\mathrm{M}_1^{\mathrm{ji}},\cdots,\mathrm{M}_\mathrm{t}^{\mathrm{ji}}$ triangle-nodes. Within $\mathrm{S_{ji}}$, for each $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$ (r=1,$\cdots$,t), for which both $\mathrm{C}_\mathrm{r}^{\mathrm{ji,left}}$ and $\mathrm{C}_\mathrm{r}^{\mathrm{ji,right}}$ exist, in order to generate the local invariant $\mathrm{G}_\mathrm{r}^{\mathrm{ji}}$ and the assumption $\mathrm{A}_\mathrm{r}^{\mathrm{ji}}$ we apply the backward approach described above by rule (2). For each $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$, for which $\mathrm{C}_\mathrm{r}^{\mathrm{ji,right}}$ does not exist, its assumption $\mathrm{A}_\mathrm{r}^{\mathrm{ji}}$ is calculated accordingly to rule (3). Its guarantee is calculated by considering the operational semantics of $\bullet$ and the guarantee of $\mathrm{M_j}$. For each $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$, for which $\mathrm{C}_\mathrm{r}^{\mathrm{ji,left}}$ does not exist, both its assumption and its guarantee are calculated accordingly to rule (1). Now, the assumption of $\mathrm{M_j}$ can be calculated by considering the assumptions of all $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$, for which $\mathrm{C}_\mathrm{r}^{\mathrm{ji,left}}$ does not exist, together with the operational semantics of $\bullet$. Intuitively, all $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$ that have no $\mathrm{C}_\mathrm{r}^{\mathrm{ji,left}}$ and all $\mathrm{M}_\mathrm{r}^{\mathrm{ji}}$ that have no $\mathrm{C}_\mathrm{r}^{\mathrm{ji,right}}$ represent the frontier between two levels of the abstract tree hierarchy. The relationship between these levels is defined by the operational semantics of the $\bullet$ operator.

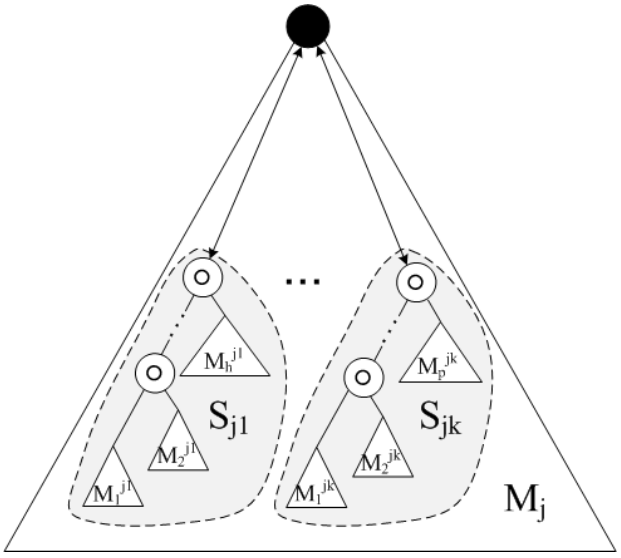

Figure 3. Subtree hierarchy of $\mathrm{M_j}$

As an example, let us assume that in $\mathrm{M_j}$ the $\bullet$ operator is an **if-then-else-fi** statement. Thus $\mathrm{M_j}$ has two subparts $\mathrm{S_{j1}}$=$\mathrm{M}_1^{\mathrm{j1}}\circ\cdots\circ\mathrm{M}_\mathrm{h}^{\mathrm{j1}}$ and $\mathrm{S_{j2}}$=$\mathrm{M}_1^{\mathrm{j2}}\circ\cdots\circ\mathrm{M}_\mathrm{p}^{\mathrm{j2}}$ corresponding to the body of the **then** and **else** branches, respectively. The assumption of $\mathrm{M}_1^{\mathrm{j1}}$ and $\mathrm{M}_1^{\mathrm{j2}}$, and the guarantees of $\mathrm{M}_\mathrm{h}^{\mathrm{j1}}$ and $\mathrm{M}_\mathrm{p}^{\mathrm{j2}}$, are calculated through the operational semantics of **if-then-else-fi**. That is, the guarantee of both $\mathrm{M}_\mathrm{h}^{\mathrm{j1}}$ and $\mathrm{M}_\mathrm{p}^{\mathrm{j2}}$ are exactly the same as the guarantee of $\mathrm{M_j}$ as it is directly implied by the operational semantics of **if-then-else-fi**. Furthermore, $\mathrm{M}_1^{\mathrm{j1}}$ is preliminarily annotated with the minimal assumption that makes the **if**-guard evaluated to true. Analogously, $\mathrm{M}_1^{\mathrm{j2}}$ has attached a preliminary minimal assumption that makes the **if**-guard evaluated to false. These assumptions are preliminary since they reflect only the semantics concerning the evaluation of the **if**-guard. By following our approach and considering the body of the **then** and **else** branches, they will be refined in order to take into account the evaluation of the entire **if-then-else-fi** statement. The assumption on $\mathrm{M_j}$ will be the "or" of the two refined assumptions on the branches as implied by the operational semantics of **if-then-else-fi**. See Section 4.1 for further details on the example.

4. Illustrative examples

In this section we make use of two illustrative examples at two different levels of abstraction in order to show informally the soundness of the framework. The first example, see Section 4.1, concerns a simple programming language whereas the second one, see Section 4.2, concerns software architecture description and verification. Given a system S, an invariant I, the composition operator $\circ$, and the operational semantics of the language, we recall that the purpose of the framework is to decorate the syntax abstract tree with the assumptions that the system components have to satisfy, and what they locally guarantee, in order to make S satisfying I.

4.1. A simple programming language

We consider a simple[2] imperative programming language whose expressions can be defined by the following grammar.

Syntax of expressions: Exp **language**

```
e::= v | c | e + e | e - e | e == e
c::= n | b
n::= d | nd
b::= true | false
d::= 0|1|2|3|4|5|6|7|8|9
v::= x|y|z|...
```

The terms generated by e (Exp) are called expressions. A value $\underline{e}$ (Val) can be associated to an expression. This value depends on the value $\underline{v}$ of the variables v (Var) contained in e. To define the operational semantics of expressions we have to consider the concept of system's state. A state σ ($State$) can be considered as a function $\sigma : Var \rightarrow Val$. Note that σ can be partially defined over the domain Var. In the following we denote with σ,σ',σ'',$\cdots$,σ_1,σ_2,$\cdots$ generic states. We also denote with $\sigma[\underline{v}/\mathtt{v}]$ the state σ in which the value of v has been replaced by $\underline{v}$.

The operators to build an expression are summation (+), subtraction (-), and logical equality (==).

The operational semantics exp is a set of inference rules defining a relation $D \subseteq Exp \times State \times Val$. The relation D is the least relation satisfying the rules. Hereafter if $(e, \sigma, \underline{v}) \in D$, we write $\langle e, \sigma \rangle \rightarrow_{exp} \underline{v}$. The rules defining exp are:

Operational Semantics of Exp

$$\langle\, \mathtt{c}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{c} \quad (exp_{const})$$

$$\langle\, \mathtt{v}\,, \sigma \,\rangle \longrightarrow_{exp} \sigma(\mathtt{v}) = \underline{v} \quad (exp_{var})$$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n} \quad \langle\, \mathtt{e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n}' \quad \underline{v} = \underline{n} + \underline{n}'}{\langle\, \mathtt{e+e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{v}} \; (exp_{\mathtt{+}})$$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n} \quad \langle\, \mathtt{e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n}' \quad \underline{v} = \underline{n} - \underline{n}'}{\langle\, \mathtt{e\text{-}e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{v}} \; (exp_{\mathtt{-}})$$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n} \quad \langle\, \mathtt{e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{n}' \quad \underline{v} = (\underline{n} == \underline{n}')}{\langle\, \mathtt{e==e}'\,, \sigma \,\rangle \longrightarrow_{exp} \underline{v}} \; (exp_{\mathtt{==}})$$

A program in our programming language can be defined by the following grammar:

Syntax of programs: $Prog$ **language**

```
p::= v:=e | p;p | if e then p else p fi
```

The terms generated by p ($Prog$) are called programs. The operators to build a program are program sequencing (**;**), variable assignment (**:=**), and conditional control (**if-then-else-fi**).

The operational semantics $prog$ is a set of inference rules defining a relation $R \subseteq Prog \times State \times State$. The relation R is the least relation satisfying the rules. Hereafter if $(p, \sigma, \sigma') \in R$, we write $\langle p, \sigma \rangle \rightarrow_{prog} \sigma'$. The rules defining $prog$ are:

Operational Semantics of $Prog$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{v}}{\langle\, \mathtt{v:=e}\,, \sigma \,\rangle \longrightarrow_{prog} \sigma[\underline{v}/\mathtt{v}]} \; (prog_{\mathtt{:=}})$$

$$\frac{\langle\, \mathtt{p}\,, \sigma \,\rangle \longrightarrow_{prog} \sigma_1 \quad \langle\, \mathtt{p}'\,, \sigma_1 \,\rangle \longrightarrow_{prog} \sigma'}{\langle\, \mathtt{p;p}'\,, \sigma \,\rangle \longrightarrow_{prog} \sigma'} \; (prog_{\mathtt{;}})$$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{true} \quad \langle\, \mathtt{p}\,, \sigma \,\rangle \longrightarrow_{prog} \sigma'}{\langle\, \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{p}\ \mathbf{else}\ \mathtt{p}'\ \mathbf{fi}\,, \sigma \,\rangle \longrightarrow_{prog} \sigma'} \; (prog_{\mathbf{if}_t})$$

$$\frac{\langle\, \mathtt{e}\,, \sigma \,\rangle \longrightarrow_{exp} \underline{false} \quad \langle\, \mathtt{p}'\,, \sigma \,\rangle \longrightarrow_{prog} \sigma'}{\langle\, \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{p}\ \mathbf{else}\ \mathtt{p}'\ \mathbf{fi}\,, \sigma \,\rangle \longrightarrow_{prog} \sigma'} \; (prog_{\mathbf{if}_f})$$

Let us consider the following program p:

```
x:=10;
if x==0 then y:=x+1 else y:=x-1 fi;
z:=x-y
```

By parsing p, its abstract syntax tree can be automatically built as shown in Figure 4. In this example the abstract composition operator $\circ$ is instantiated to the sequencing operator **;**. We choose **;** because, as shown by the definition of $prog$, it is the operator used to compose programs in

[2]For the sake of simplicity we do not consider issues concerning data type declarations, type checking, structured blocks, and scoping. Note that these issues are not relevant for the purposes of the example.

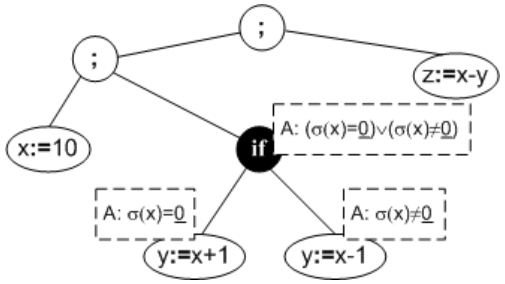

Figure 4. Abstract syntax tree of p

order to obtain a new program and it enjoys the associativity property. The two assignment-nodes concerning the value of the variables x and z, are triangle-nodes that do not have subtrees. The **if**-filled-node is a triangle-node as well, but it has two subtrees corresponding to the abstract syntax trees of the **then** and **else** branches, respectively. Furthermore, the subtree concerning the **if** triangle-node is annotated with preliminary assumptions automatically inferred from the $prog_{\textbf{if}}$ inference rules. In particular, by reading the $prog_{\textbf{if}_t}$ rule we can see that the **then** branch is enabled when the assumption $\sigma(x) = \underline{0}$ holds. Similarly, by reading the $prog_{\textbf{if}_f}$ rule we can see that the **else** branch is enabled when the assumption $\sigma(x) \neq \underline{0}$ holds. Consequently, the **if**-filled-node is annotated with the logical disjunction of the annotations of the **then** and **else** branches.

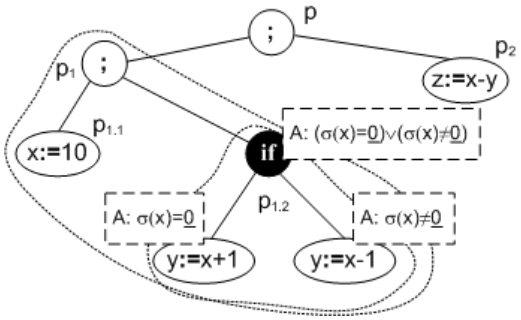

Figure 5. Decomposition of p

According to the **;** operator, p can be decomposed in the subprograms $\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{p}_{1.1}$, and $\mathtt{p}_{1.2}$, as shown in Figure 5. Let us assume that the invariant of p is $\sigma(z) = \underline{1}$ (note that this invariant is arbitrarily chosen). As it is shown in Figure 6, the invariant represents the first annotation that our framework allows the developer to attach as guarantee (G) of the root node of p. The assumption (A) for the root node of p, for now, is automatically set to $\underline{true}$. At the end of our assumption generation process, it may be further constrained (i.e., from $\underline{true}$ to some Boolean predicate over state's variables).

According to the operational semantics of the **;** operator (i.e., to the inference rule $prog_{\textbf{;}}$), our framework moves on $\mathtt{p}_2$. Here, according to the strategy of the assume-guarantee reasoning (see Section 3), the guarantee is the same of p and the assumption is automatically inferred from the $prog_{\textbf{:=}}$ rule. Since the invariant assesses that the variable z must be always evaluated to $\underline{1}$, the assignment implementing $\mathtt{p}_2$ implies that $x - y$ should be always evaluated to $\underline{1}$, hence leading to the generated assumption for $\mathtt{p}_2$. By referring to [14], to generate the assumptions we use a TAS solver for classical propositional logic. TAS denotes a family of refutational satisfiability testers for both classical and non-classical logics which, like tableaux methods, also builds models for non-valid formulas. The basis of the methodology is the alternative application of reduction strategies over formulas and a branching rule. The included reduction strategies are based on equivalence or equisatisfiability transformations. When no more simplifications can be applied, then the branching strategy is used and then the simplifications are called for.

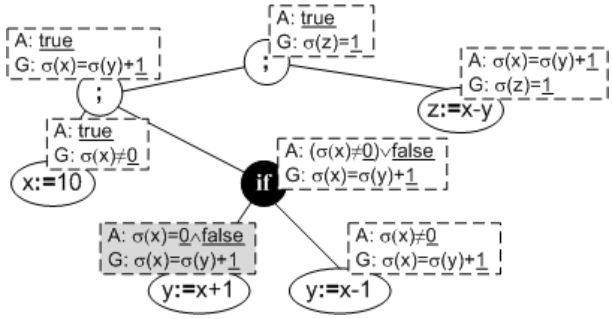

Figure 6. Assume-guarantee annotations of p

Now, our framework moves on $\mathtt{p}_1$. Here, as defined by the assume-guarantee strategy, the guarantee of $\mathtt{p}_1$ has to be the same as the assumption for $\mathtt{p}_2$ (since p is the composition of $\mathtt{p}_1$ and $\mathtt{p}_2$). The assumption for $\mathtt{p}_1$, for now, can be the same as the assumption for p.

Then, our framework moves on $\mathtt{p}_{1.2}$. Analogously to what has been done for p and $\mathtt{p}_2$, the guarantee of $\mathtt{p}_{1.2}$ is the same as the one of $\mathtt{p}_1$. Since $\mathtt{p}_{1.2}$ is a triangle-node with subtrees, in order to refine its initial assumption (i.e., $\sigma(x)=\underline{0} \vee \sigma(x)\neq\underline{0}$), our framework first moves on its right-hand subtree. Here, the guarantee is the same as the one of $\mathtt{p}_{1.2}$ and the initial assumption (i.e., $\sigma(x)\neq\underline{0}$) is not further refined since the guarantee trivially holds from the assignment y:=x-1. Then, the framework moves on the left-hand subtree of $\mathtt{p}_{1.2}$. Here the assumption is refined, from $\sigma(x)=\underline{0}$ to $\sigma(x)=\underline{0} \wedge \underline{false}$ (i.e., $\underline{false}$), since the guarantee does not hold (see the assignment y:=x+1).

Finally, our framework backtracks to $\mathtt{p}_{1.2}$ and its assumption is refined from $\sigma(x)=\underline{0} \vee \sigma(x)\neq\underline{0}$ to $\sigma(x)\neq\underline{0} \vee \underline{false}$ (i.e., $\sigma(x)\neq\underline{0}$). Concluding, our framework moves

on $p_{1.1}$ by backtracking to p_1. Here, the guarantee has to be the same as the one of $p_{1.2}$ and since it trivially holds from the assignment x:=10, the assumption is $\underline{true}$. Our assumption generation process ends by backtracking to p whose assumption $\underline{true}$ does not need to be refined.

At the end, as it is shown in Figure 6, we have automatically annotated each node of the syntax tree of our program with the assumptions on the context, in that point of the program, that are required to satisfy the considered invariant.

The pairs of assumptions and guarantees shown in Figure 6 can be rewritten in the assume-guarantee reasoning as follows:

$\langle\rangle\ p\ \langle\sigma(z) = \underline{1}\rangle$
$\langle\sigma(x) = \sigma(y) + \underline{1}\rangle\ p_2\ \langle\sigma(z) = \underline{1}\rangle$
$\langle\rangle\ p_1\ \langle\sigma(x) = \sigma(y) + \underline{1}\rangle$
$\langle\sigma(x) \neq \underline{0}\rangle\ p_{1.2}\ \langle\sigma(x) = \sigma(y) + \underline{1}\rangle$
$\langle\rangle\ p_{1.1}\ \langle\sigma(x) \neq \underline{0}\rangle$

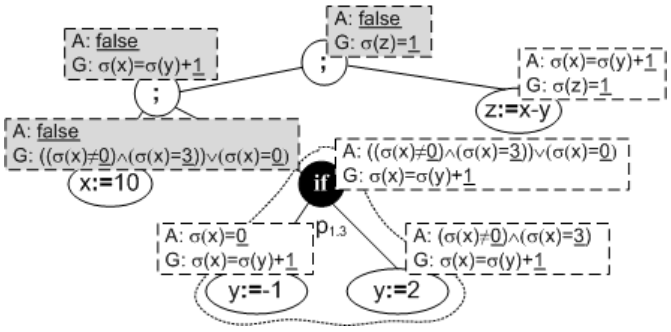

Figure 7. Assume-guarantee re-computation after adaptation

Now, let us assume that an adaptation is needed and in particular that $p_{1.2}$ (see Figure 4) is substituted by $p_{1.3}$ (see Figure 7). The new program is as follows:

```
x:=10;
if x==0 then y:=-1 else y:=2 fi;
z:=x-y
```

As we can see in Figure 7, $p_{1.3}$ has an assumption on the context that is different from $p_{1.2}$ since now both the **then** and **else** branches can concur in the satisfaction of the program invariant $\sigma(z) = 1$. In order to deal with this situation the assumptions and guarantees of each part of $C^{\text{left}}_{p_{1.3}}$ must be updated (i.e., the assumptions and guarantees of $p_{1.1}$ shown in Figure 5). Due to this update, other possible assumption updates have to be propagated to the root of the abstract syntax tree. In particular, in this case no context exists for the program able to satisfy the program invariant. The problem is on $p_{1.1}$ that assigns 10 to the variable x, whereas its guarantee is $\sigma(x)=\underline{3} \vee \sigma(x)=\underline{0}$. Therefore, for this case, there can be two possible ways to proceed: (i) $p_{1.3}$ cannot be used, (ii) $p_{1.1}$ that represents the context of $p_{1.3}$ should change. Indeed, in general, it might be also that the generated assumption for p_2, and hence the guarantee of p_1, would be too strong in the sense that it characterizes a very constraining (more than necessary) context. As we will show later in Section 4.2, this case may occur.

In the following we choose the second way to proceed. Then, the new program is as follows:

```
if x==0 then y:=-1 else y:=2 fi;
z:=x-y
```

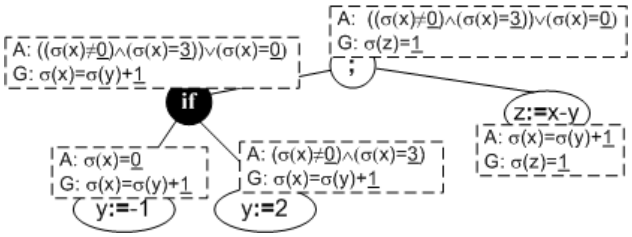

Figure 8. Contextual assumption computation

The new annotated abstract tree is represented in Figure 8. The new generated assumptions and guarantees show that this new version of p cannot be used in any context. In fact, the constraint $\sigma(x)=\underline{3} \vee \sigma(x)=\underline{0}$ is imposed on the context in which p can be used. In other words, this constraint is an abstract characterization of the possible contexts (programs) in which p can be used (as a sub-program). Note that, as an example, we used a very simple programming language in which we do not have taken into account *loops* such as a **while-do-endw** statement. In this case, possible solutions can be applied [7]. They calculate (under a suitable approximation) an upper bound on the number of times a **while-do-endw** is performed.

4.2. Analysis of software architectures specified by the FSP notation

We consider now the software architecture description of a system that is a simple communication channel, *Channel*, composed of two components: *Input* and *Output*. This description is borrowed by [5]. The language that we use to describe the system is FSP (Finite State Process notation) [13]. FSP is a language that enables modeling behavioral aspects of a software system in terms of concurrent processes. Each automaton produced via an FSP is an LTS (Labeled Transition System). Two fundamental elements are present in an FSP specification: actions (which make up processes) and states. Actions are either input actions that come from the process's environment, or output actions that originate from the process. Processes can be described recursively. By convention, action names are lower-case and process names are upper-case. For a comprehensive

description of the syntax and operational semantics of FSP, we refer to [13]. In the following we report the FSP description of the *Channel* system.

```
Input=(input->send->ack->Input).
Output=(send->output->ack->Output).
||Channel=(Input||Output).
```

The *Input* component receives an input when the `input` action occurs and then sends it to the *Output* component with action `send`. After some data is sent to it, *Output* produces output (action `output`) and acknowledges that it has finished (action `ack`). At this point, the composed system *Channel* returns to its initial state (and, hence, its components *Input* and *Output* too) and a new execution of the system can be repeated.

Figure 9. Abstract syntax tree of *Channel*

The abstract syntax tree of *Channel* is represented in Figure 9. The selected composition operator is the parallel composition (i.e., `||`) of FSP. As described above, *Channel* is obtained by composing in parallel the *Input* and *Output* components. The properties are expressed in FSP and the invariant property `I` is as follows:

```
I=(input->output->I).
```

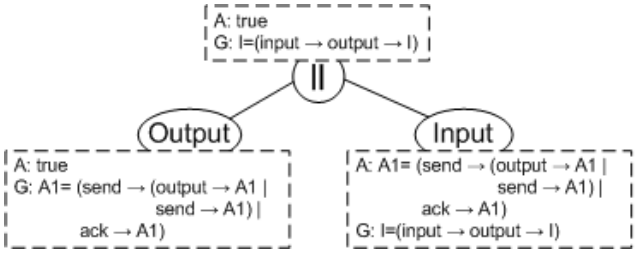

Figure 10. Assume-guarantee annotations of *Channel*

Figure 10 shows the abstract syntax tree of *Channel*, annotated with assume and guarantee properties.

By following the approach presented in Section 3, `I` is the guarantee of both the root node and of the *Input* node. In order to guarantee `I`, the *Input* node requires the assumption `A1` that is obtained by executing the $\mathcal{L}^*$ approach as described in [5]. Informally, $\mathcal{L}^*$ learns a unknown regular language and produces an automaton that accepts it. In order to do that, $\mathcal{L}^*$ needs to interact with a *Teacher*. The Teacher must be able to answer a *membership query* and a *conjecture*. The former consists in establishing whether a trace of the automaton being built is a string of the language to be learned. If the answer to the latter is true then the final automaton is built. Otherwise the Teacher returns a counterexample. The counterexample can be used to strengthen the generated assumption when it is too weak, i.e., it does not restrict the environment enough for the invariant to be satisfied. See [5] for details.

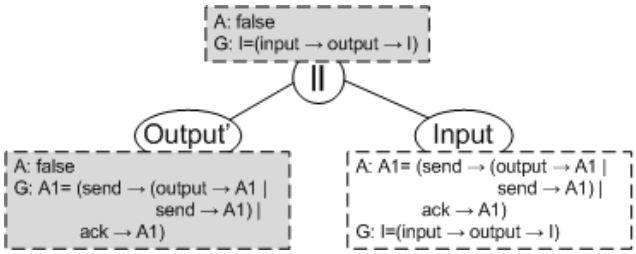

Figure 11. Assume-guarantee annotations after adaptation

`A1` becomes the guarantee of the *Output* component that is able to guarantee this property without assumptions on its context. Let us assume now that we want to substitute the *Output* component with this new component *Output′*:

```
Output'=(send->(send->Output'|
                output->ack->Output')).
```

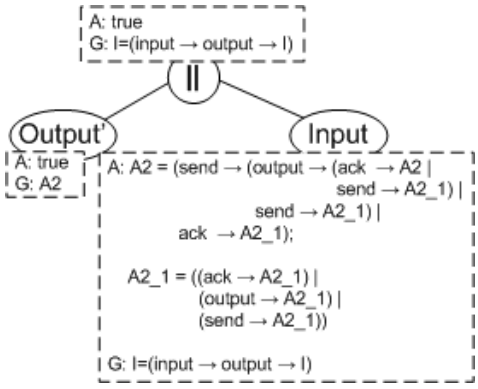

Figure 12. Assume-guarantee annotations after assumption refinement

As it is shown in Figure 11, by maintaining the guarantee `A1`, no context exists in which *Output′* can satisfy that guarantee (i.e., the assumption of *Output′* is false). Thus, differently from the example described in Section 4.1 in which we have chosen to change the context, here, we can either raise an unsuccessful notification (since the change does not keep the invariant) or check whether `A1` is too strong, and hence refine it.

The problem is that multiple `send` actions can occur before producing `output`. By following the approach in [5], we can use the counterexample produced by the verification of $\langle true \rangle$ *Output'* $\langle A1 \rangle$, i.e., $\langle$`send,send,output`$\rangle$, in order to refine `A1`. Then, $\mathcal{L}^*$ produces the weaker assumption `A2` that is shown in Figure 12.

5. Related work

Many works have been recently proposed in compositional verification and in particular in assume-guarantee reasoning. Focusing only on the most recent ones we can refer to [8, 5] and to [6]. In the first group of papers the authors present a novel framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. Their work applies to concurrent systems modeled through finite state automata and, hence, it works at a high-level of abstraction. Instead, the second approach focuses on source code verification, hence working at a lower abstraction level. In both cases the aim of these works is to provide some kind of automation. This is the key issue in assume-guarantee reasoning because its practical impact has been limited so far due to the non-trivial human input required. Each of these works can be considered as a particular instance of the framework that we have preliminarily described in this position paper.

Our framework shares ideas and motivations with the domain of *rely-guarantee reasoning* [11, 12] that concerns the compositional verification of shared-variable concurrent programs. In this domain, our framework would represent the way to automatically compute rely conditions that usually are hard to be specified.

The work in [9] is somehow related to our work and presents a run-time monitoring and verification technique able to assure that the adaptive software system satisfies its requirements. The application context and the goal of this work is deeply different from ours. Instead of using an assume-guarantee approach, the authors define a new model checker able to check both standard temporal logic properties (actually limited to safety properties) and adaptive properties. The state explosion problem is mitigated since each verification is triggered by the monitoring and then performed on a single execution trace.

The authors of [4] report experiments on how to *break up* a system into subsystems with the purpose of efficiently verifying system properties. They compared assume-guarantee verification and monolithic verification by experimenting with several systems. In order to effectively apply assume-guarantee, the authors experimented different system decompositions and they discovered that assume-guarantee verification is successful in very few cases. In their study starting from all possible two-way decompositions for a set of systems and properties, they scale the best solutions to n-way decompositions. The two-way decompositions demonstrated that approximatively half of the decompositions were better than the monolithic verification. Instead the n-way decompositions did not confirm this good result. The reason leading to this odd result is not clear and ultimately it might also depend on the used verification technique and on the underlying composition operator. In our opinion, the experimentation described in [4] despite its conclusions, still motivates the utility of a framework that assists the assume-guarantee reasoning. Indeed, it is true that the assumptions generation is one of the most expensive task of the assume-guarantee reasoning but their study is based only on the $\mathcal{L}^*$ algorithm. Moreover, the very different results of the two-way and n-way decompositions might depend on the proposed generalization.

6. Discussion

This position paper is a first step towards the definition of a theoretical assume-guarantee framework to define efficient conditions to be proved at run-time for guaranteeing the correctness of the adaptation of a composed system `S`. To this aim the framework helps in breaking a system `S` in parts whose "adaptation" can be checked in order to show that the invariant property is not hurt. The framework is based on the syntactic representation of a system and can be applied at different levels of abstraction spanning from code to software architecture. However this framework cannot be considered a panacea. As testified also by [4], breaking up is hard to do and depends on many aspects. For this reason, the framework identifies four main dimensions to be considered when dealing with assume-guarantee reasoning:

Composition operator: the composition operator should be carefully selected and must be associative.

Assumptions generation: a suitable assumption generation technique should be defined. The selection of an efficient technique is fundamental in order to assure a successful assume-guarantee reasoning.

System and property decomposition: the composition operator defines a system decomposition. The property verification should be decomposable and more precisely the assumption generation should be able, for each $\mathrm{M_x}$, to generate, starting from its guarantee properties, assumptions on its context such that $\mathrm{M_x}$ composed with its context satisfies its guarantees.

Languages selection: one of the main characteristics of our approach is that it is syntactic. This means that the role played by the language used to write the system and by the language used to specify the assumption/guarantee properties is crucial. Obviously, in this context, establishing a semantic relationship between these two languages (when they are not the same) is fundamental to make the assumption generation process fully automatic.

As discussed in Section 1, in the context of (self−)adaptable systems, our framework can play an important role in different application scenarios: (i) to ease the task of effectively breaking up the system; (ii) to ease the task of correctly (with respect to the invariant) compose a system out of elementary components; (iii) to adapt a system at different levels of granularity and at different stages, i.e., before and after deployment; (iv) compositional assumption generation; and (v) to support reactions to unsuccessful (with respect to the invariant) adaptations.

7. Future work

As next steps we plan to fully formalize the framework in order to give a proof of its soundness and provide a basis for its automation. A rigorous formalization shall allow us to gain evidence that on specific systems applying a change and verifying its correctness locally (with respect to the changed sub-system) provides an advantage over checking the invariant against the entire system. The implementation of the framework will permit to experiment it on case studies that belong to different application domains and that specify adaptation at different granularity. This would allow us to assess the generality of our framework and establish whether the set of identified dimensions for an effective application of assume-guarantee reasoning is complete. Finally, testing our theory in real situations would allow us to understand its contribution and its limitation.

Acknowledgments

This work is partly supported by both the CONNECT (Emergent Connectors for Eternal Software Intensive Networked Systems) and the d-ASAP (Software Architectures for Dependable and Adaptable Pervasive Computing Systems: design, analysis, and validation) projects. CONNECT is part-funded by the EU under the 7th Framework Program, IST priority Contract No. 231167 - http://www-rocq.inria.fr/arles/Connect/index.html. d-ASAP is an Italian PRIN 2008-2010 project.

References

[1] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE 2004)*. ACM Press, 2004.

[2] S. Chaki and N. Sinha. Assume-guarantee reasoning for deadlock. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 134–144, Washington, DC, USA, 2006. IEEE Computer Society.

[3] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos et al. 08031 – software engineering for self-adaptive systems: A research road map. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, number 08031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[4] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–52, 2008.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning Assumptions for Compositional Verification. In *Ninth International Conference on Tools and Algorithm for the Construction and Analysis of Systems (TACAS 2003)*, number 2619 in LNCS, Warsaw, Poland, April 2003. Springer.

[6] J. Dingel. Computer-Assisted Assume/Guarantee Reasoning with VeriSoft. In *25th International Conference on Software Engineering (ICSE2003)*, Portland, Oregon, May 2003.

[7] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[8] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. In *Automated Software Engineering journal, 12(3): 297-320*, 2005.

[9] H. J. Goldsby, B. H. Cheng, and J. Zhang. AMOEBA-RT: Run-Time Verification of Adaptive Software. *Models in Software Engineering: Workshops and Symposia at MoDELS 2007*, pages 212–224, 2008.

[10] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[11] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[12] C. B. Jones, C. Morgan, and S. Verlag. Wanted: a compositional approach to concurrency. In *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[13] J. Kramer and J. Magee. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

[14] M. Ojeda-Aciego and A. Valverde. TASCPL: TAS Solver for Classical Propositional Logic. In *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 738–741. Springer, 2004.

[15] A. Pnueli. In transition for global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, 1984.